A Formal Model for Generating Integrated Functional and User Interface Test Cases

Daniel Sinnig[1], Ferhat Khendek[2], Patrice Chalin[2]

[1]Department of Software Engineering
Institute of Computer Science
University of Rostock, Germany
dasin@informatik.uni-rostock.de

[2]Faculty of Engineering and Computer Science
Concordia University, Montreal
Quebec, Canada
{khendek, chalin}@encs.concordia.ca

Abstract— **Black box testing focuses on the core functionality of the system, while user interface testing is concerned with details of user interactions. Functional and user interface test cases are usually generated from two distinct system models, one for the functionality and one for the user interface. As a result, test cases derived from either model capture only partial system behavior and as such, are inadequate for testing full system behavior. We propose a method for formally integrating the model for the system functionality and the model for the user interface. The resulting composite model is then used to generate more complete test cases, capturing detailed user interactions as well as secondary system interactions. In this paper we employ use cases for modeling system functionality, and task models for describing user interfaces.**

Keywords- Functional Testing, User Interface Testing, Use Case Models, Task Models, Labeled Transition Systems

I. INTRODUCTION

Use case models are the medium of choice for expressing functional requirements, while task models are employed to capture User Interface (UI) requirements and design information. As such, use case models capture UI independent primary interactions as well as interactions with secondary actors and internal system steps. Task models capture UI specific primary interactions but leave out secondary interactions and internal computations. Both models are used to derive functional test cases [1;2;3] and UI test cases [4;5;6], respectively.

Since both the use case model and the task model capture only a partial view of the system, test cases derived from either artifact only cover a subset of the system's interactions, omitting either the detailed user interactions or the interactions with secondary actors. Consequently, due to the lack of a comprehensive testing model, functional testing and UI testing are often performed separately in a nonintegrated manner; i.e., the functionality is typically tested without the UI and the UI is tested without the system functionality.

A running system, however, is the result of the integration of both aspects and as such should ideally be tested in an integrated manner. This requires an adequate model of the expected system behavior against which the behavior of the running system can be compared. If possible this model should already be in use in the development process or be derivable from already existing models. In this paper, we propose the development of such a composite model from use case and task models. Test cases generated from this composite model cover both aspects of the system in an integrated manner.

We define a common formal semantics for use case and task models based on the well-known formalism of labeled transition systems (LTSs). The semantics enables us to define a refinement relation between the two artifacts, which formally captures the notion of consistency between them. Furthermore, under the assumption that the task model is a valid refinement of the use case model, we formally define the merging of both artifacts to a composite LTS. Finally we explain how this unified model can be used to generate integrated test cases using existing techniques [7;8;9;10]. To our knowledge this is the first formal technique proposed for generating such integrated tests. The research results presented here build upon and extend the foundational work on a core semantics for use cases and task models described in [11;12].

The remainder of this paper is structured as follows: In Section 2, we provide background information on use case and task models and relate both artifacts from a practical viewpoint. Section 3 presents mathematical preliminaries surrounding LTSs needed for the definition of a common formal semantics. The latter is given in Section 4, together with a formalization of refinement. Based on the formal semantics we define the integration of both artifacts in Section 5 and its usage for test case generation. Section 6 reviews relevant related work. We conclude in Section 7.

II. BACKGROUND

In this section we provide the necessary background information on use case and task modeling. For each concept we present the main features, concrete notations, and a comprehensive example. Finally, both concepts are compared, and main commonalities and differences are contrasted. The insights thus gained allow us to formally integrate use cases and task models at the semantic level.

A. Use Case Modeling

Since their introduction by Jacobson [13] in the early 90s, creating use cases has become a key software development activity. Growing evidence suggests that the adoption of use case modeling leads to significant benefits

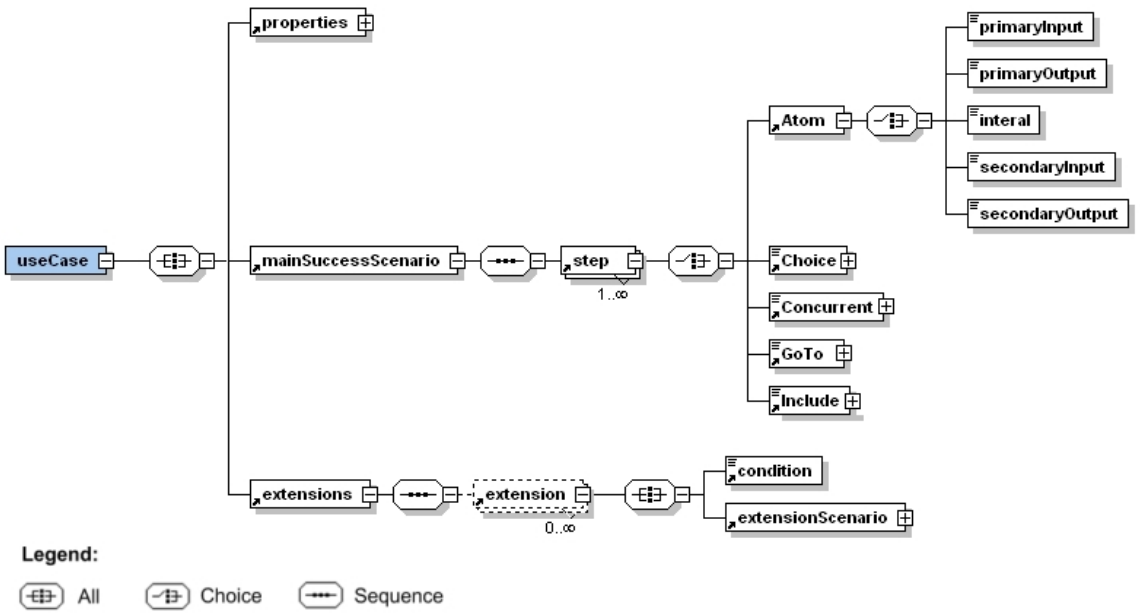

Figure 1. Use Case Model Structure

for customers and developers alike, through clearer requirements understanding, and through greater programming automation [14;15;16].

A use case model captures the "complete" set of use cases for an application, where each use case specifies possible usage scenarios for a particular functionality offered by the system. Every use case starts with a header section consisting of various properties (e.g., primary actor, goal, goal-level, etc.). The core of a use case is its main success scenario, which lays out the most common way for the primary actor to reach their goal when using the system. Use case extensions define alternative scenarios which may or may not lead to fulfillment of the use case goal.

Different notations for expressing use cases possessing varying degrees of formality have been suggested. They range from purely textual constructs written in prose [14] to entirely formal specifications written in Z [17], as well as abstract state machines [18] and graph structures [19]. In this article we adopt an intermediate solution [20;21], which enforces a formal structure, but also preserves the intuitive nature of use cases: We formalize the sequencing of use case steps and their kinds and types, but leave the associated actions and conditions to be specified informally.

As shown in Figure 1, a use case consists of a property section, a main success scenario, and a set of extensions. Both the main success scenario as well as each extension consist of a sequence of use case steps, which can be of five different kinds: *Atomic* steps are performed either by an actor (primary or secondary) or the system, and contain no sub-steps; *Choice* steps provide the primary actor with a choice between several interactions, each interaction being in turn defined by a sequence of steps; *Concurrent* steps define a set of steps the primary actor may perform in any order; *Goto* steps denote jumps to other steps within the same use case; and finally, *Include* steps define the inclusion of a sub-use case.

Atomic steps are further distinguished by the type of interaction they describe. In particular, we distinguish between *input* (?), *output* (!), and *internal* (~) steps. Input steps are either carried out by the primary actor (*primary input*) or a secondary actor (*secondary input*). Output steps denote system messages that are either directed to the

Use case: Order Product

Properties

Primary Actor: Customer
Secondary Actor: Payment System
Goal: Primary Actor places an order for a specific product.
Goal-Level: User-goal

Main Success Scenario

1. Primary Actor specifies desired product category. [?[1] spCA]
2. System displays products matching category criteria. [![1] diSR]
3. Primary Actor selects a product in desired quantity. [?[1] slPQ]
4. System verifies product's availability in requested amount. [~ vaPQ]
5. System presents the purchase summary. [![1] diPS]
6. Primary Actor submits payment information. [?[1] prPI]
7. System requests Payment System to process the payment. [![2] rqPA]
8. Payment System confirms the payment. [?[2] coPA]
9. System provides confirmation number to Prim. Actor. [![1] prCN]

Use case ends successfully

Extensions

4a. Desired product is not available:
4a1. System informs Primary Actor of unavailability. [![1] inPU]
Use case ends successfully.

8a. The Payment was not authorized.
8a1. Payment System indicates payment is not authorized [?[2] caPA]
8a2. System informs Prim. Actor payment is not authorized. [![1] inPD]
Use case ends unsuccessfully.

Figure 2. "Order Product" Use Case

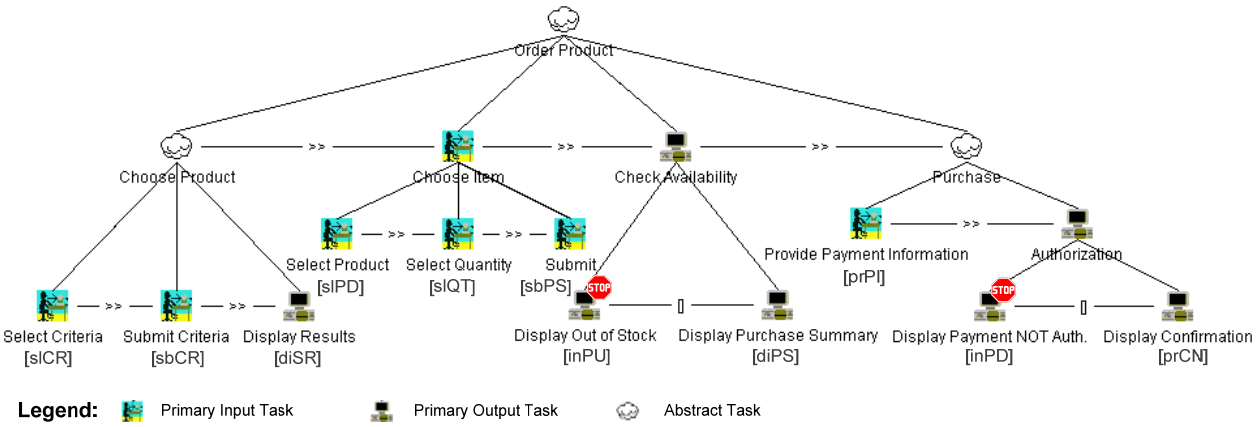

Figure 3. "Order Product" Task Model

primary actor (*primary output*) or a secondary actor (*secondary output*). *Internal* steps denote internal system computations which do not yield a visible effect, for neither the primary nor for any secondary actor.

A use case model is well-formed if: (1) Every use case in the use case model, except for a designated root use case, is directly or indirectly included in the root use case. (2) Each internal system step is associated with an extension and is followed by an output step. (3) Any sequence of internal and/or secondary interaction steps is directly preceded by a primary input step, and is directly followed by a primary output step. For the full list of well-formedness rules please consult [11].

The use case of Figure 2 captures the interactions for the "Order Product" functionality of an invoicing system. The main success scenario describes the situation in which the primary actor directly accomplishes their goal of ordering a product. The two extensions specify alternative scenarios, which both lead to the abandonment of the use case goal. For convenience, each use case step has been further attributed with a mnemonic label and a type, where $?^1$ and $?^2$ denote primary input and secondary input steps respectively, $!^1$ and $!^2$ denote primary output or secondary output steps, and ~ denotes an internal step.

B. Task Modeling

Task modeling is a well accepted technique supporting user-centered UI design [5]. In most UI development approaches, the task model is the primary input to the UI design stage. *Task models* capture the complete set of tasks that users are allowed to perform when using the application, as well as how the tasks are related to each other. Like use cases, task models describe the user's interaction with the system (primary interactions). Task models however, do not capture secondary interactions and internal system steps, as these are irrelevant for UI design [21]. The primary purpose of task models is to systematically capture the way users achieve a goal when interacting with the system [22].

Various notations for task models exist. Among the most popular ones are ConcurTaskTrees (CTT) [5], GOMS [23], TaO Spec [24], and HTA [25]. Even though all these notations differ in terms of presentation, level of formality, and expressiveness, they share the following common tenet: Tasks are hierarchically decomposed into sub-tasks until an atomic level has been reached. In what follows we describe in detail the *Extended ConcurTaskTrees* (*ECTT*) task-modeling notation first introduced in [26], which is an extension of CTT.

In ECTT, tasks are arranged hierarchically with more complex tasks decomposed into simpler sub-tasks. ECTT includes a set of binary and unary temporal operators: The former are used to temporally link sibling tasks at the same level of decomposition, whereas the latter are used to qualify *optional*, *iterative*, and *stop* tasks. A summary of ECTT operators is given in Table 1. We note that the binary operators have similar—yet not semantically identical—counterparts in LOTOS [27].

TABLE 1. ECTT TEMPORAL OPERATORS

Operator	Syntax	Interpretation
Enabling	$t_1 \gg t_2$	Upon termination of t_1, t_2 becomes enabled.
Choice	$t_1 [\,] t_2$	Either t_1 or t_2 is executed. The execution of one task disables the other one.
Order Independence	$t_1 \boxplus t_2$	Execution of t_1 and t_2 in any order.
Concurrency	$t_1 \parallel t_2$	Interleaved execution of t_1 and t_2 and their subtasks
Iteration	t^*	t may be executed repetitively (0 or many times).
Optional Tasks	$[t]$	t may be executed or not.
Stop	stop(t)	t cannot enable any tasks

An example of an ECTT task model is given in Figure 3. It corresponds to the "Order Product" use case defined in Figure 2. The task model is visualized as a task tree, which clearly portrays the hierarchical breakdown of high-level tasks into lower-level tasks. The execution order of tasks is determined by the temporal operators that are defined between peer tasks. An indication of the task type is given by the icon used to represent a task. ECTT distinguishes

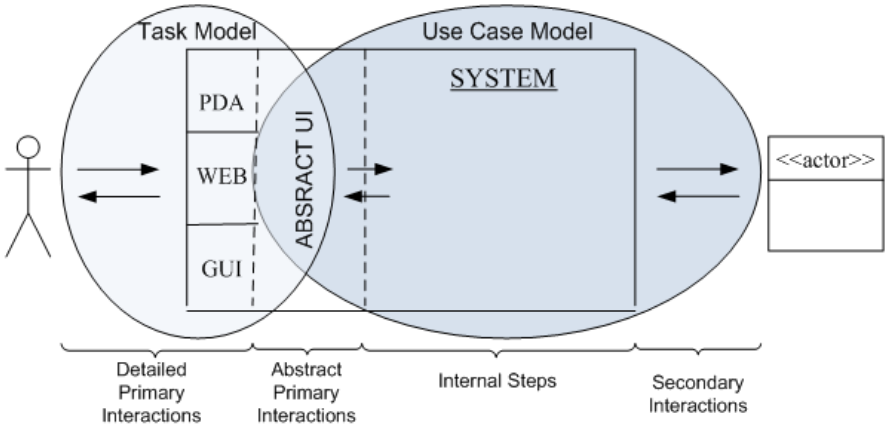

Figure 4. Scope of Actions in Use Case and Task Models

between three different task types: *primary input*, *primary output*, and *abstract* tasks. Primary input tasks are performed by the user (through the UI), primary output tasks are performed by the system and have an externally visible outcome to the user. Abstract tasks denote high-level tasks which can involve both primary input and primary output tasks. As a well-formedness criterion for ECTT task models, we require that the task type of atomic tasks be either primary input or primary output.

C. Use Case and Task Models in the Development Process

In practice, use case models are employed to document functional requirements, whereas task models are used to describe UI requirements and designs. Ideally, the functional requirements captured by use cases are independent of any particular UI. In contrast, the requirements and design information captured by task models take into account the specifics of a particular type of UI. The upshot is that the use case model captures the bare functional requirements of the system, which are then further "refined" by the task model, taking UI specific interactions into account. With this understanding we identify two main differences that are pertinent to *how* both models are applied by practitioners:

In use case models, requirements are captured at a higher-level of abstraction while task models are more detailed. Hence, the atomic actions of a task model are often lower-level UI details that are irrelevant (actually contraindicated [14]) in the context of a use case.

Task models concentrate on aspects that are relevant for UI design. As such, usage scenarios are strictly depicted as input-output relations between the user and the system. Internal steps and secondary interactions—which are hidden from the end user—as specified in use case models, are not captured.

As depicted in Figure 4, viewed separately, the use case and task model capture only a partial view of the system. As a consequence, test suites derived from either artifact will never capture the full system behavior. More comprehensive test suites can be obtained if both models are integrated into a single composite model, which can then be used for test case generation.

In what follows we discuss how such integration is possible, under the condition that the task model is a *refinement* of the use case model, i.e., both are consistent models of the same system. A prerequisite for the definition of the refinement relation and the integration of use case and task models is a common formal semantics; the definition of which is detailed in the subsequent sections.

III. Notation and Mathematical Preliminaries

The common semantic domain we propose for use case and task models is a *typed labeled transition system* (tLTS). Its definition is similar to the definition of an ordinary LTS [28] with the exception that each action is assigned a unique type. Moreover, we characterize a subset of the states as final states in order to use them in the sequential composition of tLTSs.

Definition 1. Typed Labeled Transition System. A *typed labeled transition system* is a tuple $(Q, L, \longrightarrow, q_0, F, \rho)$ where Q is a countable nonempty set of states, L is a set of observable actions, $\longrightarrow \subseteq Q \times (L \cup \{\tau\}) \times Q$ is the transition relation (where τ denotes the distinguished unobservable action, such that $\notin L$), and $q_0 \in Q$ is the initial state. We extend the notion of traditional LTSs by reserving a set of final states $F \subseteq Q$ for the sequential composition (given in Section IV) of tLTSs. The typing function $\rho: L \longrightarrow \{?^1, !^1, ?^2, !^2\}$ associates each element in L with a unique type reflecting the various use case step types and task types.

We use $q \xrightarrow{a} q'$ to denote that $(q, a, q') \in \longrightarrow$. For graphic representation of an tLTS $(Q, L, \longrightarrow, q_0, F, \rho)$ we use an arrow to indicate the initial state and a double-circle to denote final states in F. Furthermore, the standard abbreviations defined in Figure 5 will be used as shorthand.

Let $A \subseteq L; q, q' \in Q; \sigma = a_1 a_2 \dots a_n; a \in L; u \in (L \cup \{\tau\})$ ***then***		
$q \xrightarrow{u} q'$	$\Leftrightarrow$	$\exists q'. q \xrightarrow{u} q'$
$q \xrightarrow{\sigma} q'$	$\Leftrightarrow$	$\exists q_1, \dots, q_{n-1}. q \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_{n-1}} q_{n-1} \xrightarrow{a_n} q'$
$q \overset{a}{\Longrightarrow} q'$	$\Leftrightarrow$	$q \xrightarrow{\tau^* a \tau^*} q'$
$q \overset{\sigma}{\Longrightarrow} q'$	$\Leftrightarrow$	$\exists q_1, \dots, q_{n-1}. q \overset{a_1}{\Longrightarrow} q_1 \overset{a_2}{\Longrightarrow} \dots \overset{a_{n-1}}{\Longrightarrow} q_{n-1} \overset{a_n}{\Longrightarrow} q'$
$q \overset{a}{\Longrightarrow}$	$\Leftrightarrow$	$\exists q'. q \overset{a}{\Longrightarrow} q'$
$q \overset{\sigma}{\Longrightarrow}$	$\Leftrightarrow$	$\exists q'. q \overset{\sigma}{\Longrightarrow} q'$
q ref A	$\Leftrightarrow$	$\nexists a \in A. q \overset{a}{\Longrightarrow}$

Figure 5. Notations for tLTSs

In what follows we remind the reader of relevant definitions pertinent to labeled transition systems [7;28]. The *set of all traces* of a tLTS M is defined as: $traces(M) = \left\{\sigma \in L^* \mid q_0 \overset{\sigma}{\Longrightarrow}\right\}$. Each trace contains a sequence of observable actions. We also define the set of traces that lead to a particular state q in M : $tracesTo(M, q) = \left\{\sigma \in L^* \mid q_0 \overset{\sigma}{\Longrightarrow} q\right\}$. q after $\sigma = \left\{q' \mid q \overset{\sigma}{\Longrightarrow} q'\right\}$ denotes the set of all states reachable from q by accepting the sequence σ. The set of refusals for a given state q and sequence of actions σ is defined as: $\text{Ref}(q, \sigma) = \cup A. \exists q' \in (q \text{ after } \sigma). q' \; ref \; A$. Having specified the notions of *traces* and *refusals*, we are now able to define *testing equivalence*. Two systems are testing equivalent, if in addition to trace equivalence, they have the same refusal properties [7].

Definition 2. Testing Equivalence between tLTSs. Let $M_1 = (Q_1, L_1, \rightarrow_1, q_{0_1}, F_1, \rho_1)$, $M_2 = (Q_2, L_2, \rightarrow_2, q_{0_2}, F_2, \rho_2)$ be two tLTSs. M_1 and M_2 are *testing equivalent* ($M_1 \equiv_{te} M_2$) iff: $traces(M_1) = traces(M_2)$ and

$$\forall \sigma \in traces(M_1). Ref(q_{0_1}, \sigma) = Ref(q_{0_2}, \sigma).$$

Note that this notion of equivalence does not take into account the respective final states sets (F_1 and F_2) and typing functions (ρ_1 and ρ_2) as these will only be needed for the composition and merging of tLTSs.

Finally we define the set of unstable states, which is required for the integration procedure presented in Section 5. A state is unstable if it serves as a source state for at least one transition triggered by τ.

Definition 3. Unstable States. The set of *unstable states* $Q^\tau \subseteq Q$ is defined as follows: $Q^\tau = \left\{q \mid \exists q'. q \xrightarrow{\tau} q'\right\}$.

IV. A Common LTS Semantics for Use Case and Task Models

A prerequisite for the behavioral merge of use case and task model is a common formal semantics, and a definition of refinement that relates both artifacts. An overview of both will be given in this section. The full details can be found in [11].

A. Semantics for Use Case Models

The semantic mapping from a use case model into a tLTS is defined in a bottom-up manner, starting with the mapping of individual use case steps. Each of the five kinds of use case steps enumerated in Figure 1 has its own specific mapping to a tLTS. We start with the mapping of *atomic* steps. As shown in Figure 6, depending on the step type, two different tLTSs are generated. The rationale behind each case is as follows:

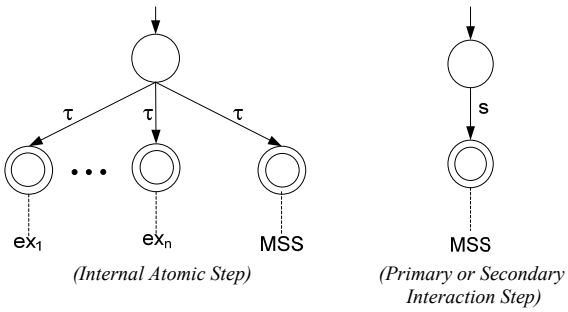

Figure 6. Mapping of Atomic Use Case Steps

Internal use case steps are not observable by actors and thus may non-deterministically result in $n + 1$ different outcomes, one being captured in the main success scenario, and the remaining n outcomes in the corresponding extensions. As depicted in Figure 6 (left) the former (MSS) results in a final state, which will be used for the sequential composition with the tLTS representing the next step in the main success scenario. The latter $(ex_1, \dots, ex_n)$ result in a set of states which will be used for the sequential composition with the tLTSs representing the various extension steps.

In contrast to internal system steps, which are performed by the system and cannot be observed by actors, primary or secondary interactions are performed or observed by either the primary or a secondary actor. As such, they do not have an alternative outcome *per se*, but may be associated (by virtue of one or more extensions) with alternative steps which are performed *instead* of the actual step. The resulting tLTS consists of only one transition, representing the use case step (Figure 6 right). Alternative steps are captured in the tLTSs representing the corresponding extensions.

The mappings of the remaining step kinds are briefly outlined next. A *Choice* step is mapped to a composite tLTS which unifies the initial states of each possibility's tLTS (Figure 9 right). A *Concurrent* step corresponds to the construction of the product machine of its constituent tLTSs. A *Goto* step is mapped to a tLTS consisting of a single state defined to be equivalent to the initial state of the jump target's tLTS. The corresponding tLTS for an *Include* step consists of two states: one identified with the initial state of the tLTS of the main success scenario of the invoked sub-use case, and the other identified with all final states of the sub-use case's tLTS.

Now that individual use case steps can be formally represented by tLTSs, we can link arbitrary sequences of steps using *sequential tLTS composition* (Figure 9 left),

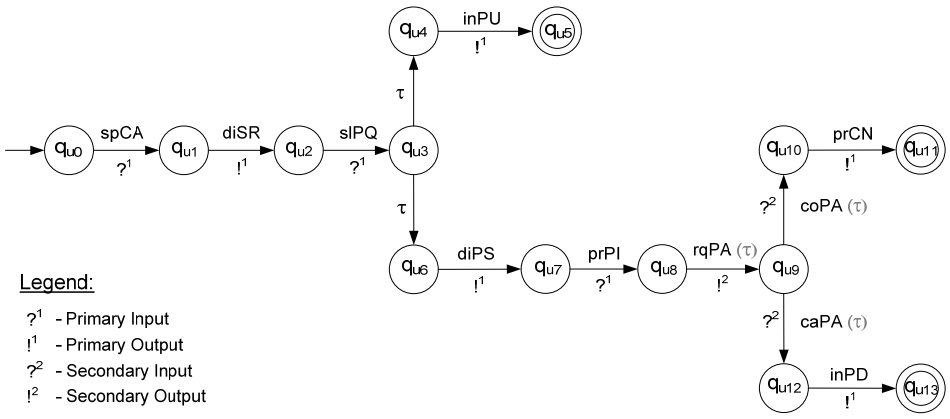

Figure 7. tLTS Representation of "Order Product" Use Case

which unifies the final states of its first operand with the initial state of its second operand. An entire use case is mapped to a tLTS by relating the tLTSs representing the main success scenario and all extensions. Similarly, an entire use case model is mapped to a tLTS by composing the tLTSs representing its use cases, which, in accordance with well-formedness rule (1) (Section II.A) are interrelated through *include* relationships. Figure 7 portrays the tLTS obtained from the semantic mapping of the "Order Product" use case of Figure 2. It has three final states: q_{u11} denoting the successful outcome of the use case (i.e., the customer's order placement succeeded), q_{u5} denoting the case where the product is not available, and q_{u13} denoting the case where the payment was not authorized.

B. Semantics for Task Models

Similar to the semantic mapping of use cases, the mapping of ECTT task models to tLTSs is performed in a bottom-up manner. Building blocks are the *atomic* tLTS, $skip$ tLTS, and $stop$ tLTS as depicted in Figure 8. Each atomic ECTT task expression is mapped into an atomic tLTS, whose single action adopts the same type as the associated atomic task (i.e., primary input ($?^1$) or primary output ($!^1$)).

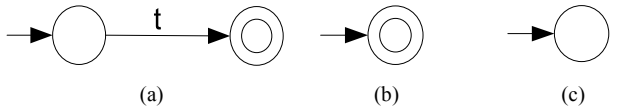

Figure 8. (a) Atomic tLTS (t_{LTS}), (b) $skip$ LTS and (c) $stop$ LTS

Composite ECTT task expressions are represented by more complex tLTSs, which result from the composition of the tLTSs representing sub-expressions. For this purpose, we have defined the following composition operations: *sequential composition* ($\cdot$), *deterministic choice composition* ($\#_D$), *nondeterministic choice composition* ($\#_N$), *parallel composition* ($\|$), and *iterative composition* ($*$). In what follows, we will give an overview of these operations. The full details can be found in [11]. As already discussed in the previous section, the sequential composition of tLTSs consists of unifying the final states of the first operand and the initial state of the second operand. The deterministic choice composition merges the initial states of the involved tLTSs. Both compositions are schematically depicted in Figure 9.

In contrast to deterministic choice composition, nondeterministic choice composition does not merge the initial states of the involved tLTSs, but instead introduces a new initial state which contains τ transitions to the initial states of the respective operant tLTSs. The parallel composition of two tLTSs constructs a product machine that defines all possible "interleavings" of the transitions defined in both operands. The iterative composition defines for each state in F, a new transition to each of the states directly reachable from q_0. Using the composition operations we define the semantic mapping from ECTT task expressions to tLTS as follows:

Definition 4. Mapping an ECTT Task Expression to tLTS. Let t be an atomic ECTT task and υ, φ be arbitrary ECTT task expressions. We then define the mapping $\mathcal{M}$ to tLTSs as follows:

$$\mathcal{M}[\![t]\!] = t_{tLTS}$$

$$\mathcal{M}[\![\upsilon \gg \varphi]\!] = \mathcal{M}[\![\upsilon]\!] \cdot \mathcal{M}[\![\varphi]\!]$$

Figure 9. *Sequential* (left) and *Deterministic Choice Composition* (right)

$$\mathcal{M}[\![v\ [\ ]\ \varphi]\!] = \begin{cases} \mathcal{M}[\![v]\!] \#_D \mathcal{M}[\![\varphi]\!], & if\ \rho(v) = \rho(\varphi)\ = ?^1 \\ \mathcal{M}[\![v]\!] \#_N \mathcal{M}[\![\varphi]\!], & if\ \rho(v) = \rho(\varphi) = !^1 \end{cases}$$

$$\mathcal{M}[\![v\ \|\ \varphi]\!] = \mathcal{M}[\![v]\!]\ \|\ \mathcal{M}[\![\varphi]\!]$$

$$\mathcal{M}[\![v\ \boxplus\ \varphi]\!] = \mathcal{M}[\![(v \gg \varphi)\ [\ ]\ (\varphi \gg v)]\!]$$

$$\mathcal{M}[\![[v]]\!] = \mathcal{M}[\![v]\!] \#_D\ skip$$

$$\mathcal{M}[\![v^*]\!] = (\mathcal{M}[\![v]\!])^*$$

$$\mathcal{M}[\![stop(v)]\!] = \mathcal{M}[\![v]\!] \cdot stop$$

Noteworthy is the semantic mapping of the ECTT choice operator ([ ]). If both operands are of type primary output ($!^1$) (e.g., "Display Success Message", "Display Failure Message") then the nondeterministic choice ($\#_N$) is used to compose the corresponding tLTSs. Otherwise, if both operands are of type primary input ($?^1$) the deterministic choice composition ($\#_D$) is employed. This correlates with the assumption of Section II.C that task models do not capture interactions that are invisible to the end user. A direct consequence of this "under-specification" is that all system choices appear to be nondeterministic.

As an example for the semantic mapping, Figure 10 depicts the tLTS representing the "Order Product" task model. There are three final states, among which two (q_{t8}, q_{t13}) represent the premature abortion of the task model. One final state (q_{t15}) represents the successful completion of the task model.

C. Refinement Relation

In this section we define a refinement relation between use case and task models. For a refinement to be valid, we require that the task model only structurally refine the use case model without restricting the set of possible scenarios. Alternative refinement criteria between use case and task models (e.g., behavioral refinement through trace inclusion [21]) are also plausible, but for the sake of simplicity will not be discussed within this work. Formally, a task model is a refinement of a given use case model, if and only if, the respective tLTSs are *testing equivalent*.

Definition 5. Refinement. Let U be a use case model and T be a task model and M_U and M_T their respective tLTS representations. Then T is a refinement of U iff: $M_U \equiv_{te} M_T$

One precondition for the application of the definition is that the involved tLTSs operate over the same alphabet. In the following, we briefly discuss two techniques to resolve alphabet conflicts, called *refinement mapping* and *action hiding* [11]. A *refinement mapping* collapses a set of transitions in the task model tLTS to a single transition. This is required to compensate for the fact that since task models are more detailed than use case models, an atomic use case step may be refined by more than one atomic task in the task model. For the refinement mapping to be valid we require that a mapping only be defined between actions of the same types. For our "Order Product" example we note the following two structural refinements: (1) The atomic use case step "Primary Actor specifies desired product category" is structurally refined by the tasks "Select Criteria" and "Submit Criteria;" (2) The atomic use case step "Primary Actor selects Product in desired quantity" is refined by the tasks "Select Product," "Select Quantity" and "Submit." The task model tLTS after the refinement mapping has been applied is depicted in Figure 10, where the transitions represented by dashed lines are the result of the refinement mapping. A more elaborate discussion about the various refinement mappings possible can be found in [11].

Action hiding is applied to the use case model tLTS to abstract away from all actions that are not present in the task model tLTS. As stated in Section II.C secondary interactions which are irrelevant for UI design, are not captured by task models. Hence, in order to compare use case and task models for refinement, we replace all transitions that are triggered by such actions with τ transitions. After application of refinement mapping and action hiding to the "Order Product" task and the use case model tLTSs respectively, the verification that both tLTSs are testing equivalent is straightforward, and we conclude that the "Order Product" task model is a refinement of the "Order Product" use case.

V. A Composite Model for Integrated Test Case Generation

Both the tLTS representing the "Order Product" use case and the tLTS representing the "Order Product" task model (before refinement mapping and action hiding) can be used separately to generate black box test cases. When used separately however, each generated test suite will only cover

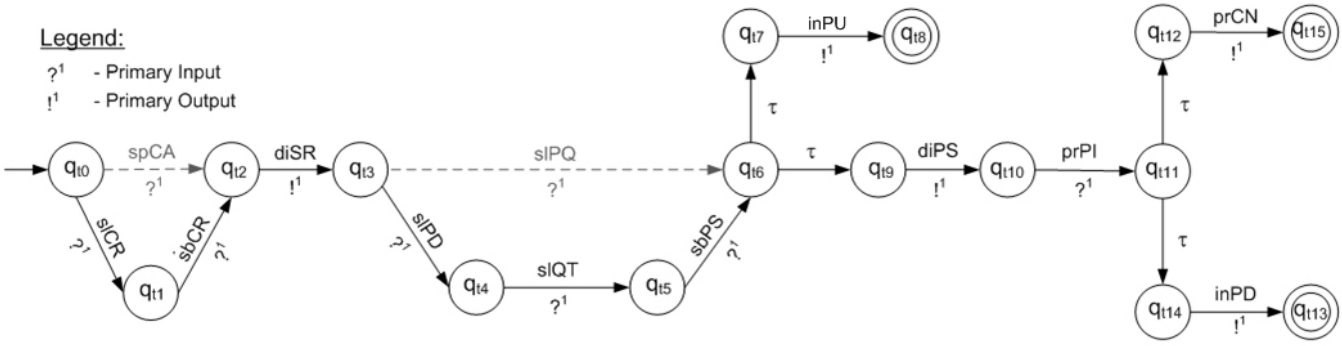

Figure 10. tLTS Representation of "Order Product" Task Model

parts of the involved interactions (i.e., test cases generated from the use case tLTS do not cover the detailed primary interactions, whereas test cases generated from the task model do not cover interactions with secondary actors). Moreover, the interworking between the functionality and UI cannot be covered. In this section, we elaborate how—under the assumption that the task model is a refinement of the use case model—both models can be *merged* into a *single composite system model* and used for the generation of integrated test cases.

We start by outlining the desired high-level behavior of the envisioned composite system model that results from application of our proposed merging procedure. Since task models are more detailed than use case models, the composite model at first adopts the interactions defined in the task model, up until a point where internal and/or secondary interactions are required. That is, whenever a primary input task (e.g., "Submit" in Figure 3) is followed by a choice between two or more primary output tasks. At this point, the composite model adopts the behavior of the use case model (in terms of internal and secondary interactions), which specifies how the user output is produced (e,g., steps 7, 8, 8a1 in Figure 2). Thereafter, the composite model again, continues with the behavior of the task model. This alternating continues until the usage scenario has come to an end.

As previously stated, a precondition for a successful merge is that the task model is a refinement of the use case model, and thus the respective tLTSs (after refinement mapping and action hiding) are testing equivalent. The composite model results from merging the tLTS representing the use case model M_U and the tLTS representing the task model M_T. The resulting tLTS M_M is defined over the union of the state sets of M_T and M_U. In order to support "context switching" between task model and use case behavior, i.e., transitions from a state in M_T to M_U and vice versa, the two state sets need to be interrelated. In what follows we introduce the auxiliary functions *TMtoUC* and *UCtoTM*. The former yields for a given state in M_T the set of trace-equivalent states in M_U. The latter is dual and returns the set of trace-equivalent states in M_T for a given state in M_U.

Definition 6. State Correspondence Mappings. Given a use case model U and a refining task model T with $M_U = (Q_U, L_U, \rightarrow_U, q_{0_U}, F_U, \rho_U)$, $M_T = (Q_T, L_T, \rightarrow_T, q_{0_T}, F_T, \rho_T)$ as their respective tLTS representations. Furthermore, let $M'_U = (Q'_U, L'_U, \rightarrow'_U, q'_{0_U}, F'_U, \rho'_U)$, $M'_T = (Q'_T, L'_T, \rightarrow'_T, q'_{0_T}, F'_T, \rho'_T)$ be the results of refinement mappings and action hiding (Section IV.C) applied to M_U and M_T respectively, such that the original state sets are preserved. We define the functions $TMtoUC$ and $TMtoUC$ as follows:

$$TMtoUC(q) = \left\{\hat{q} \in Q'_U \mid \exists \sigma \in tracesTo(M'_T, q).\, q'_{0_U} \stackrel{\sigma}{\Longrightarrow} \hat{q}\right\}$$

$$UCtoTM(q) = \left\{\hat{q} \in Q'_T \mid \exists \sigma \in tracesTo(M'_U, q).\, q'_{0_T} \stackrel{\sigma}{\Longrightarrow} \hat{q}\right\}$$

We illustrate the application of *TMtoUC* and *UCtoTM* and relate the state sets of the use case tLTS and task model tLTS of Figure 7 and Figure 10. For example, when applied to state q_{t6} in the task model tLTS, *TMtoUC* yields the set $\{q_{u3}\}$ in the use case tLTS. Similarly, if we apply *UCtoTM* to state q_{u4} of the use case tLTS, we obtain the set $\{q_{t6}, q_{t7}, q_{t9}\}$ in the task model tLTS. Note that if *UCtoTM* is applied to q_{u6} the same result is obtained. This is a direct consequence of action hiding. The definitions of *TMtoUC* and *UCtoTM* enable us to formally define the *merging* of use case and task model tLTSs. Without loss of generality, we assume that the state sets of the use case tLTS and task model tLTS are disjoint.

Definition 7. Merging. Given a use case model U and a refining task model T with $M_U = (Q_U, L_U, \rightarrow_U, q_{0_U}, F_U, \rho_U)$ and $M_T = (Q_T, L_T, \rightarrow_T, q_{0_T}, F_T, \rho_T)$ as their respective tLTS representations. We define the function $merge$ as follows: $merge(M_U, M_T) = (Q_U \cup \mathrm{Q_T}, \mathrm{L_U} \cup \mathrm{L_T}, \rightarrow_M, q_{0_T}, F_T, \rho_U \cup \rho_T)$, where the transition relation $\rightarrow_M$ is defined as:

$\forall q, q' \in (Q_U \cup \mathrm{Q_T})\ \forall u \in (\mathrm{L_U} \cup \mathrm{L_T}).\, q \stackrel{u}{\longrightarrow}_M q'$ *if and only if*

$$\left(q \stackrel{u}{\longrightarrow}_T q' \wedge q \notin Q_T^{\tau}\right) \vee \quad \text{(i)}$$

$$\left(q \in Q_T^{\tau} \wedge \exists \hat{q} \in TMtoUC(q).\, \hat{q} \stackrel{u}{\longrightarrow}_U q'\right) \vee \quad \text{(ii)}$$

$$\left(q \stackrel{u}{\longrightarrow}_U q' \wedge u \neq \tau \rightarrow \rho(u) \in \{?^2, !^2\}\right) \vee \quad \text{(iii)}$$

$$\left(q \in Q_U \wedge \rho(u) = !^1 \wedge \exists \hat{q} \in UCtoTM(q).\, \hat{q} \stackrel{u}{\longrightarrow}_T q'\right) \quad \text{(iv)}$$

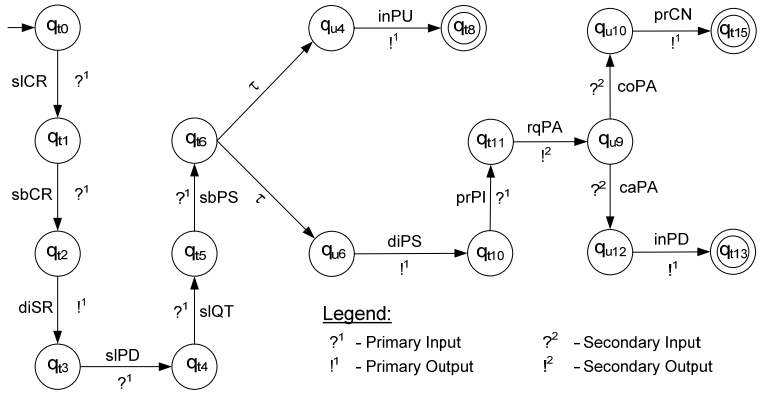

Figure 11. Composite tLTS representing the "Order Product" Use Case and Task Model

The resulting composite tLTS is defined over the union of states and actions of the respective use case and task model tLTSs. The typing function is computed in a similar way. The initial state and the final states are identified with the corresponding counterparts of the task model tLTS. The definition of the new transition relation $\rightarrow_M$ distinguishes between the following four cases; (i) Transitions originating from the state set of the task model tLTS (except for unstable states) remain as specified by M_T; (ii) Transitions originating from an unstable state in M_T are identified with the transitions originating from corresponding state in M_U. Note that if the task model is a refinement of the use case model there is only *one* corresponding state; (iii) Transitions originating from states in M_U, if triggered by τ or by actions representing secondary interactions, remain as specified in M_U; (iv). Transitions originating from a state in M_U triggered by a primary output interaction are identified with the corresponding transition from a trace equivalent state in M_T.

We note that the definition of the transition relation takes advantage of the following: whenever the task model tLTS is in an unstable state ($q \in Q_T^\tau$) the corresponding task model expects the choice of two or more primary output tasks (see Definition 4), and hence a "context switch" ($TMtoUC$) to the state set of the use case model tLTS is needed. Similarly, whenever the use case tLTS is ready to perform a primary output action ($!^1$), the sequence of internal steps and/or secondary interactions in the corresponding use case has ended and a "context switch" ($UCtoTM$) back to the task model state space is defined. We note that according to well-formedness rule (3) for use case models, a secondary interaction can never be followed by a primary input interaction. The merge of the use case model tLTS (Figure 7) and the task model tLTS (Figure 10) results in the tLTS depicted in Figure 11 (only reachable states are shown).

Once the composite tLTS has been generated, it can be used to further drive development of the system. In our case, we are interested in using this model for testing the system, and the integration of functionality and UI. Test cases generated from this model will cover the detailed primary interactions (as specified in the task model), as well as secondary interactions (as specified in the use case model). Test case generation from LTSs has been extensively studied in the literature and any one of several test case generation techniques and strategies available can be used [7;8;9;10].

VI. Related Work

To our knowledge, this is the first attempt to propose a formal integration of use case and task models for the purpose of deriving test cases.

In the literature, test cases are typically derived in a *nonintegrated* manner either from use case models or from task models. Test case generation from use cases has been proposed by various researchers. Nebut et al. [3] proposes a formal method for deriving both functional and robustness test cases from parameterized use cases, where pre- and post-conditions for use case execution are formalized in OCL. Froehlich and Link [2] describe a manual approach to generating system-level test cases from use cases, including pre-conditions and post-conditions. Each use case is transformed into a state machine which is then used for test case generation based on certain test objectives defined by the tester. In [1], Briand and Labiche describe a method for the derivation of test cases and test oracles from use case diagrams, use case descriptions, and class diagram. Kassel [29] presents a methodology to generate test cases from structured use cases. Similar to our approach, in a structured use case only the sequencing of use case steps is formalized, while the respective actions, as well as associated conditions are specified informally.

The literature has identified task models as a suitable means for generating user interface test cases [5]. The approach presented by Benz [4] uses CTT task models to describe the interaction between users and the system. Based on a formal interpretation of the CTT temporal operators, test cases are derived directly from the task model. The authors also define a set of task model specific coverage criteria such as task coverage, temporal relationship coverage, and random selection. Silva et al. [6] present an approach which uses CTT task models as test oracles for model-based testing of user interfaces. Similar to our approach, task models are transformed into a finite state machine which serves as a basis for test case generation. Testing is facilitated by the "Spec Explorer" tool which automates the execution of test cases.

A merge algorithm for (partial) scenario-based specifications (modal transition systems) is described by Uchitel and Chechik [30]. As a precondition for their approach, it is required that both models describe consistent views of the system. Similar to our approach, alphabet conflicts are addressed through event hiding. The merge is performed through parallel composition of the operand models while synchronizing on shared actions.

VII. Conclusion

In this paper, we proposed a method to create an integrated system model for the generation of functional and user interface test cases. The integration is obtained by merging use case and task models. The former are the specification medium of choice for functional requirements, while task models are employed to capture UI requirements and design information. The common semantics of our tLTS formalism forms the basis of the behavioral merge. On their own, the use case and task model tLTSs can be used to generate separate functional or user interface test cases. These test cases however, capture only partial system behavior, which either omits the detailed user interactions, or the interactions with secondary actors. Moreover, the integration of these two aspects is ignored by such test cases.

We have shown that under the assumption that the task model is a refinement of the use case model, i.e., both models are consistent, the artifacts can be combined to obtain a more accurate model of the running system. This model integrates the internal computations and secondary interactions defined in the use case model, with the detailed primary interactions of the task model. Technically, the merge is performed by combining the state sets of tLTSs representing both artifacts in such a way that the resulting tLTS is alternatively either in a task model state or in a use

case state. The resulting composite tLTS can then be used to generate more complete and concrete test cases covering the detailed user interactions and interactions with secondary actors.

As future work, we plan to carrying out more comprehensive case studies and applying our integration procedure to real work examples involving use case models and task models. Other avenues also include an extension of the proposed semantics to capture state and data information, which is often employed in use case or task models to express and evaluate conditions. This will likely require the use of Extended Finite State Machine (EFSMs) and corresponding test case generation techniques [10]. Moreover, we envision that the semantic mappings as well as the merging procedure be aided by supporting tools, supporting the non-trivial tasks of authoring, validating, and merging of use case and task models.

ACKNOWLEDGMENT

This work is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) in the form of a Postgraduate Scholarship for D. Sinnig and Discovery Grants for P. Chalin and F. Khendek. We are grateful to Steven Barrett and Homa Javahery who meticulously reviewed our work.

REFERENCES

[1] Briand, L. and Labiche, Y., A UML-Based Approach to System Testing, in Proc. of *UML 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, Toronto, Canada, pp. 194 - 208, 2001.

[2] Fröhlich, P. and Link, J., Automated Test Case Generation from Dynamic Models, in Proc. of *ECOOP'00*, Sophia Antipolis and Cannes, France pp. 472-492, 2000.

[3] Nebut, C., Fleurey, F., Le Traon, Y. and Jezequel, J., Automatic Test Generation:A Use Case Driven Approach, in IEEE Trans. Softw. Eng., **32** (**3**), pp. 140-155, 2006.

[4] Benz, S., Combining Test Case Generation for Component and Integration Testing, in Proc. of *AMOST'07*, London, UK, pp. 23-33, 2007.

[5] Paternò, F., Model-Based Design and Evaluation of Interactive Applications, Springer, 2000.

[6] Silva, J., Campos, J. C. and Paiva, A., Model-based user interface testing with Spec Explorer and ConcorTaskTrees, in Proc. of *Formal Methods for Interactive Systems* Macau, China, 2007.

[7] Brinksma, E., Scollo, G. and Steenbergen, LOTOS specifications, their implementations, and their tests, in Proc. of *IFIP Workshop Protocol Specification, Testing, and Verification VI*, pp. 349-360, 1987.

[8] White, L., Almezen, H. and Sastry, S., Firewall Regression Testing of GUI Sequences and their Interactions, in Proc. of *International Conference on Software Maintenance*, Amsterdam, NL, IEEE Computer Society, pp. 398-410, 2003.

[9] Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M. and Pretschner, A., Model-Based Testing of Reactive Systems, Springer, 2005.

[10] Lee, D. and Yannakakis, Principles and Methods of Testing Finite-State Machines-A Survey, in Proceedings of IEEE, **84** (**8**), pp. 1089-1123, 1996.

[11] Sinnig, D., Use Case and Task Models: Formal Unification and Integrated Development Methodology, PhD Thesis in *Department of Computer Science and Software Engineering*, Concordia University, Montreal, 2008.

[12] Sinnig, D., Chalin, P. and Khendek, F., LTS Semantics for Use Case Models, in Proc. of *ACM - SAC 2009*, Honolulu, HI, 2009.

[13] Jacobson, I., Object-Oriented Software Engineering : A Use Case Driven Approach, ACM Press , New York, 1992.

[14] Cockburn, A., Writing Effective Use Cases, Addison-Wesley, Boston, 2001.

[15] Larman, C., Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition), Prentice Hall PTR, 2004.

[16] Selic, B., The pragmatics of model-driven development, in IEEE Software, **20** (**5**), pp. 19-25, 2003.

[17] Butler, G., Grogono, P. and Khendek, F., A Z Specification of Use Cases, in Proc. of *APSEC 1998*, pp. 94-101, 1998.

[18] Barnett, M., Grieskamp, W., Schulte, W., Tillmann, N. and Veanes, M., Validating use-cases with the AsmL test tool, in Proc. of *Quality Software 2003*, pp. 238-246, 2003.

[19] Mizouni, R., Salah, A., Kolahi, S. and Dssouli, R., Merging partial system behaviours: composition of use-case automata, in Software, IET, **1** (**4**), pp. 143-160, 2007.

[20] Sinnig, D., Chalin, P. and Khendek, F., Consistency between Task Models and Use Cases, in Proc. of *DSV-IS 2007*, Salamanca, 2007.

[21] Sinnig, D., Chalin, P. and Khendek, F., Common Semantics for Use Cases and Task Models, in Proc. of *Integrated Formal Methods*, Oxford, England, pp. 579-598, 2007.

[22] Souchon, N., Limbourg, Q. and Vanderdonckt, J., Task Modelling in Multiple contexts of Use, in Proc. of *DSV-IS*, Rostock, Germany, pp. 59-73, 2002.

[23] Card, S., Moran, T. P. and Newell, A., The Psychology of Human Computer Interaction, 1983.

[24] Dittmar, A., Forbrig, F., Stoiber, S. and Stary, C., Tool Support for Task Modelling - A Constructive Exploration, in Proc. of *DSV-IS 2004*, July 2004.

[25] Annett, J. and Duncan, K. D., Task Analysis and Training Design, in Occupational Psychology, **41**, pp. 211-221, 1967.

[26] Sinnig, D., Wurdel, M., Forbrig, P., Chalin, P. and Khendek, F., Practical Extensions for Task Models in Proc. of *TaMoDia '07*, Toulouse, France Springer, 2007.

[27] Interactions, I.-I. P. S.-O. S. (1987). ISO 8807: LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour.

[28] Keller, R., Formal Verification of Parallel Programs, in Communications of the ACM, **19**, pp. 561-572, 1976.

[29] Kassel, N., An Approach to Automate Test Case Generation from Structured Use Cases, Thesis in *Computer Science*, Clemson University, Clemson, South Carolina, 2006.

[30] Uchitel, S. and Chechik, M., Merging partial behavioural models, in SIGSOFT Softw. Eng. Notes, **29** (**6**), pp. 43-52, 2004.